BridgeBuff

In this assignment we will implement a client-server pair using remote procedure calls (RPCs) through a REST interface. In particular, we will implement a server and a client to export statistics on Bridge Defense games played in the servers in our class.

As usual for most of these systems, our solution will comprise:

1. A client program that makes REST requests, and
2. A server program that responds to client requests.

Dataset

The dataset we will use in this assignment will be provided on Canvas and contain `status` entries for completed games in the Bridge Defense servers. The `status` dictionaries contain several stats, which students will need to parse and make available over a REST API.

Using a REST API

A REST API allows systems to exchange messages through widely-available Web protocols in a platform independent fashion. In short, a REST server offers a set of URL *endpoints*, each with specific semantics, that clients can access to interact with the server. In this assignment, the different endpoints will provide different views over the dataset comprising games played in the Bridge Defense servers. Usually, the encoding used to exchange data between REST clients and servers is JSON. For example, consider the following shell command (assuming that the `wget` tool is installed and that Internet access is available):

```
wget -q https://www.peeringdb.com/api/ix/1 -O-
```

The `wget` command makes an HTTP request to the URL passed as a parameter (for full details, check the manual page for `wget`). The `https://www.peeringdb.com/api/ix/1` URL is an endpoint in PeeringDB's REST API, and the semantics associated with a request to `/api/ix/1` is to retrieve information about the Internet Exchange (`ix`) with identifier `1`. Below is a more detailed look at each component of the URL:

URL Part	Meaning
`https://www.peeringdb.com`	The REST server's IP address and TCP port

URL Part	Meaning
`/api`	Path where the REST API is hosted
`/ix`	Endpoint to manage Internet Exchange Points
`/1`	Manage the IXP with identifier `1`

The response is in JSON format, and should contain the following information (among other information):

```
{"meta": {}, "data": [{"id": 1, "org_id": 2, "org": {"id": 2, "name": "Equinix", "wel
```

A REST API can support any HTTP method. Common methods and semantics are listed in the table below, but REST servers can define and support other methods (e.g., `UPDATE`). The default method is `GET` , which usually retrieves information from the server. This is why the example command above retrieved information about IXP `1` .

HTTP/REST Method	Semantics
`GET`	Retrieve object or data
`HEAD`	Retrieve a summary of an object of data
`PUT`	Add or overwrite an object or data
`POST`	Add or update an object or data
`DELETE`	Remove an object or data

The format and information in URLs are defined by the REST API provided by the server. We call each such URL an *endpoint*.

Motivation and Examples

We will implement a REST API server for the data gathered by Bridge Defense servers. The idea behind the REST API is to provide a library to support sites like DotaBuff. These sites rely on public data exported through Valve's REST API. Some sites, like Stratz, get information from Valve's servers and provide their own REST API with derived information.

Interface Description

REST API Server

The server is responsible for loading information from the games and responding to client requests arriving at URL endpoints. On startup, the server must load the database with Bridge Defense game results that will be uploaded on Canvas. The server should implement at least three endpoints supporting the `GET` method to summarize game results.

1. `/api/game/{id}` : Retrieve information about the game with identifier `id`

   For example, a client can request information about game 10 sending a `GET` request to endpoint `/api/game/10` . The response should contain the game identifier and statistics:

   ```
   {
        "game_id": 10,
        "game_stats": {...}
   }
   ```

2. `/api/rank/sunk?limit={count}&start={index}` : Retrieve a page of games with the largest number of sunk ships

   This request receives two parameters, `limit` and `start` , which implement pagination. Pagination is necessary whenever the amount of data to send in a response grows very large. The idea is that the server will respond with a "page" of results. A response should contain (up to) `limit` entries starting by the entry number given in `start` . In our API, responses should contain no more than 50 entries; if `limit` is larger than 50, answer the request with an HTTP 400 (Bad Request) error code.

   For example, a request may ask for the top 10 games with the highest number of sunk ships sending a `GET` request to `/api/rank/sunk?limit=10&start=1` . The response should contain: (1) a `ranking` entry with the value `sunnk` to specify the type of response, (2) the `limit` used in the request, (3) the `start` index, (4) the list of `games` , and URLs for the `prev` ious and `next` page of results in the ranking:

   ```
   {
       "ranking": "sunk",
       "limit": 10,
       "start": 1,
       "games": [<int1>, <int2>, ..., <int10>],
       "prev": null,
       "next": "/api/rank/sunk?limit=10&start=11"
   }
   ```

If there is no previous or next page of results, omit the field in the response or set it to `null` .

3. `/api/rank/escaped?limit={count}&start={index}` : retrieve a page of games with the *smallest* number of escaped ships

   As in the previous endpoint, the `limit` and `start` parameters implement pagination. Requests with `limit` larger than 50 should be answered with an HTTP 400 (Bad Request) error code.

   For example, the response to a request for `/api/rank/escaped?limit=10&start=10` should contain a list of 10 games, starting with the game with the tenth lower number of escaped ships and ending with the game with the twentieth lower number of escaped ships. Responses should contain a `ranking` field with value `escaped` to indicate the type of ranking, as well as the `limit` , `start` , `games` , `prev` , and `next` fields, as defined above.

```
{
    "ranking": "escaped",
    "limit": 10,
    "start": 10,
    "games": [<int10>, <int11>, ..., <int19>],
    "prev": "/api/rank/sunk?limit=9&start=1",
    "next": "/api/rank/sunk?limit=10&start=20"
}
```

Client Program

You should also implement a command-line program to issue REST requests toward the server and perform two analyses over the data:

1. GAS with the best performance (Immortals)

   For this analysis, the client should build a CSV file with one GAS per line. Each line should contain the GAS, the number of games that GAS has in the top100 ranking of sunk ships, and the average number of ships sunk in that GAS's games in the top100. The lines should be sorted in decreasing order of games in the top100.

2. Best cannon placements (top meta)

   In this analysis the client should compute a summary of the efficacy of cannon placements. To such end, the client should retrieve the top100 ranking of games with the smallest number of escaped ships and compute the average number of escaped ships for each *normalized* cannon placement. The client should then generate a CSV file with one line per normalized cannon placement. A line should contain the normalized cannon placement and the average number of escaped ships. The lines in the CSV should be sorted by increasing number of escaped ships.

To normalize cannon placements, build a 8-digit string where the `i` -th digit contains the number of "rows" with exactly `i` cannons. The normalized cannon placement should always contain 8 digits, which may include zeros to the left. Consider the string is indexed left-to-right. (The game has 5 cannon rows, one for each river and an additional one below the fourth river.)

Implementation Details and Restrictions

The *server* can be implemented using any technology, library, and Web framework. A common choice for Python is using the Flask framework.

The *client* must *not* use any libraries to handle HTTP requests. For example, the client cannot use the `http.client` , `urllib.request` , or `requests` libraries. The *client* also must use HTTP/1.1 and *not* set the `Connection: close` header. The client should strive to reuse the TCP connection to send as many REST requests as possible.

The goal of these restrictions is to force students to get experience with (1) creating and receiving data over TCP connections as well as (2) creating and handling HTTP requests and responses. If you find your code is not handling HTTP request/responses, then you are likely using a disallowed library. One specific task that your code must handle is reading the HTTP response's `Content-Length` header to control reception of the response. Please contact the instructor for clarification when uncertain about a library.

Both servers and clients can use other libraries. For example, using a library to manipulate JSON if fine on either the server of the client.

Please do *not* include a header line in your generated CSV files. The client should be executable using the following command line:

```
./client <IP> <port> <analysis> <output>
```

Where `IP` and `port` specify the IP address and port where the server is running, `analysis` may be either 1 or 2 to specify which of the analyses above will be executed, and `output` should be a file name where the CSV file should be stored.

You can use any language to implement your server and client, and possibly different languages for each. As we are using open standards (JSON and REST), your programs should interoperate with your colleagues programs.

Grading

The first student that identifies incoherence, ambiguity, or any problem with the spec should point it out to the instructor. These students will be rewarded extra credit depending on the subtlety and impact of the reported problem.

At least the following checks will be performed:

- All endpoints on the server work correctly.
- The client implements the two analyses and generate the correct CSV files.
- The client does not use any library to handle HTTP messages.

Extra credit

The implementation of additional functionality on the server and client may be worth extra credit. Check with the instructor if you want to pursue such an idea. In particular, extending the server to provide a Web interface is worth up to 4 extra credits, and providing a documentation for the REST API is worth 1 extra credit.

Submission

Submit on Canvas the code for your REST servers and client as well as a PDF containing documentation for your project.

Your documentation should contain instructions describing how to execute your server and client. Consider that the instructor and assistants are unfamiliar with any of the technologies, libraries, and frameworks used. There is no need to explain everything, but instructions on how to execute the server are a must, and points to documentation (e.g., to the project websites) are desirable. A section on "troubleshooting" common problems would also be useful.

The documentation should also discuss why a library or framework was chosen. For example, does the chosen framework compare favorably vs alternatives? The documentation should also discuss the challenges encountered during the project and how they were addressed.